
南開大學
Nankai University

计算机学院

计算机体系结构

# 仿真实验一

姓名：赵康明

学号：210937

专业：计算机科学与技术

2023 年 9 月 30 日

# 目录

# 1 实验设计

## 1.1 asm2hex 脚本的重新编写

在本次实验过程中，首先需要将所提供的.s 汇编代码翻译转换成十六进制的机器码。而原先脚本所使用的 spim447 过于古老，且不适应目前的配置。在查询网上资料后发现使用较新版本的 spim 需要改写脚本指令，同时也发现了可更换使用其他的模拟器：Mars。Mars 是一款用 java 语言编写的适用于模拟 MIPS 汇编指令执行的硬件模拟器。于是，本次实验将 asm2hex 脚本重写如下：

asm2hex 脚本

```
#!/usr/bin/python3
import os
import sys
os.system("java -jar Mars4_5.jar " + sys.argv[1] +" dump .text HexText " +
    sys.argv[1][:-2]+".x")
}
```

脚本各参数解释如下：

- os.system(”java -jar Mars4_5.jar ” + sys.argv[1] + ” dump .text HexText ” + sys.argv[1][:-2] + ”.x”)：这是一个包含字符串拼接的 Python 表达式，用于构建要在操作系统中执行的命令。
- ”java jar Mars4_5.jar ”：这是要执行的 Java 命令的一部分。它运行了一个叫做”Mars4_5.jar” 的 Java 可执行文件，这个文件可能是 Mars 模拟器的一部分，用于执行特定的操作。
- sys.argv[1]：这是 Python 的 sys 模块中的 argv 列表，用于获取脚本的命令行参数。sys.argv[1] 表示在命令行中传递给脚本的第一个参数，在本次实验是传入的文件的名称。
- ” dump .text HexText ”：这是 Java 命令的一部分，它包含一些选项和参数，指定了 Mars 模拟器要执行的操作。具体来说：
  - ”dump”：这个选项告诉 Mars 模拟器执行”dump” 操作，通常是将某些数据或信息导出到文件。
  - ”.text”：这个参数指定要导出的数据类型，通常是程序的文本部分（代码段）。
  - ”HexText”：这个参数指定了导出数据的格式，这里指定了为十六进制文本。
  - sys.argv[1][:-2] + ”.x”：这是构建导出文件的文件名。它使用了 sys.argv[1] 的前面几个字符，然后附加了”.x” 作为文件扩展名。这个文件将包含导出的数据。

  通过这样全新的一个脚本，实现了将 MIPS 汇编代码翻译转换成十六进制的机器码。

## 1.2 sim.c 的编写完善

### 1.2.1 指令格式

在完成 sim.c 之前，我们需要掌握和学习 MIPS 指令的类型及其格式：

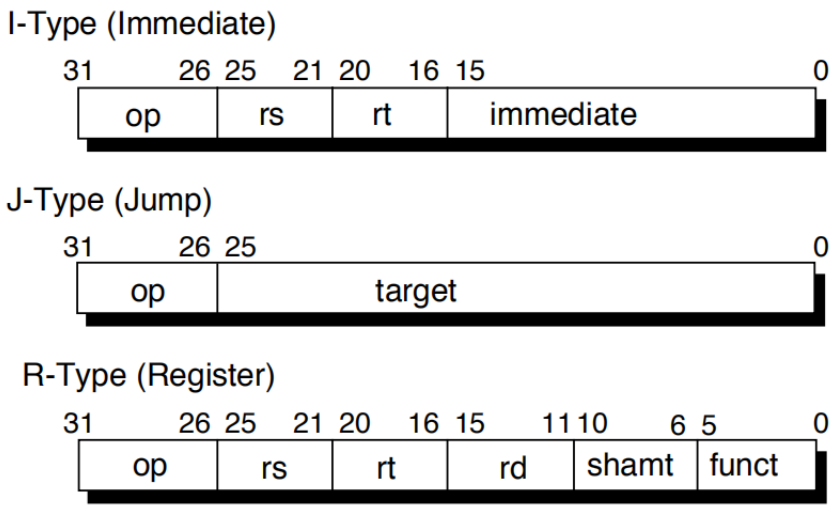



图 1.1: 指令格式

表 1: MIPS 指令格式

| 指令类型 | 字段 | 描述 |
|---|---|---|
| R-Type | 操作码 | 操作码字段，6 位 |
| | 源寄存器 1 | 源寄存器 1 标识，5 位 |
| | 源寄存器 2 | 源寄存器 2 标识，5 位 |
| | 目标寄存器 | 目标寄存器标识，5 位 |
| | 移位位数 | 移位操作的位数，5 位 |
| | 功能码 | 具体操作的功能码，6 位 |
| I-Type | 操作码 | 操作码字段，6 位 |
| | 源寄存器 | 源寄存器标识，5 位 |
| | 目标寄存器 | 目标寄存器标识，5 位 |
| | 立即数/偏移量 | 立即数或偏移量字段，16 位 |
| J-Type | 操作码 | 操作码字段，6 位 |
| | 目标地址 | 目标地址字段，26 位 |

结合 MIPS 指令格式，我们将创建取指（IF），译码（ID），执行（EXE）三个函数：

1. 取指：通过 mem_read 函数将指令从当前 PC 指针指向的内存地址取出
2. 译码：将机器指令的各个部分分段取出，如操作码 (op)、源寄存器 (rs)、目标寄存器 (rt)、目的寄存器 (rd)、移位量 (shamt)、功能码 (funct)、立即数 (imm) 和地址 (addr) 等。
3. 执行：在函数中，使用 switch 语句判断其操作码属于哪个类型的指令，执行相应的操作。对于有部分位置相同的指令，我们可以在这些小类的指令当中再进行一次 switch 语句进行分支，选择具体的操作进行执行。同时在执行的过程中，需要注意符号扩展问题。部分指令在执行的过程中，

需要将 16 位的立即数进行扩展，扩展成 32 位有符号整数，注意需保证其正负符合的正确。此外，还需要注意每次指令执行结束后，需要更新 PC 的值。最终，我们将在 Process_instructions 中调用这三个函数，完成指令执行。

# 2 实验代码

由于实验代码篇幅较长，节选部分代码进行展示：

取指和译码

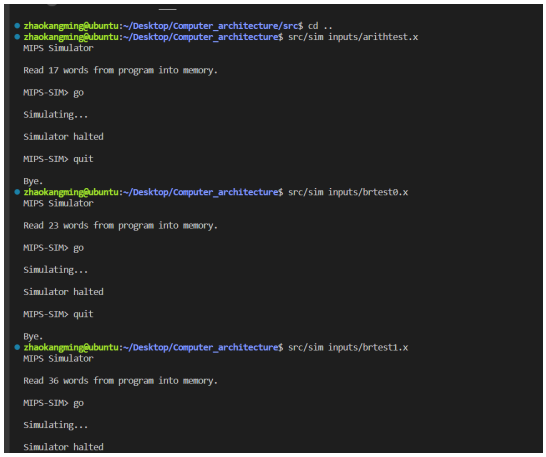
```
//取指
void IF()
{
    machine_inst=mem_read_32(CURRENT_STATE.PC);
}
//译码
void ID()
{
    //获取各个位并且将最高位清零
    op=(machine_inst>>26) & 0x3F;
    rs=(machine_inst>>21) & 0x1F;
    rt=(machine_inst>>16) & 0x1F;
    rd=(machine_inst>>11) & 0x1F;
    shamt=(machine_inst>>6) & 0x1F;
    funct=(machine_inst>>0) & 0x3F;

    jump_target=machine_inst&0x03FFFFFF;
    imm = (machine_inst >> 0) & 0x0000FFFF;
    offset =(machine_inst >> 0) & 0xFFFF;
    base=(machine_inst&0x03E00000)>>21;
}
```

执行代码部分实现

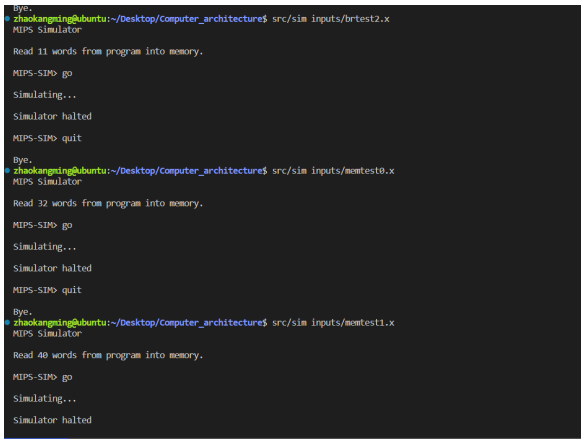
```
//取指
void IF()
{
    machine_inst=mem_read_32(CURRENT_STATE.PC);
}
 switch (op)
    {
    //1
    case OP_J:

        NEXT_STATE.PC=(CURRENT_STATE.PC&0xF0000000)|(jump_target<<2);
        break;
    //2
    case OP_JAL:
        NEXT_STATE.PC=(CURRENT_STATE.PC&0xF0000000)|(jump_target<<2);
```

```
16          NEXT_STATE.REGS[31]=CURRENT_STATE.PC+8;
17          break;
18      //3
19      case OP_BEQ:
20          branch_target=((offset<<16)>>14);
21          if(CURRENT_STATE.REGS[rs]==CURRENT_STATE.REGS[rt])
22          {
23              NEXT_STATE.PC=CURRENT_STATE.PC+branch_target;
24          }
25          else{
26              NEXT_STATE.PC=CURRENT_STATE.PC+4;
27          }
28          break;
29      //4
30      case OP_BNE:
31
32      branch_target=((offset<<16)>>14);
33      if(CURRENT_STATE.REGS[rs]!=CURRENT_STATE.REGS[rt])
34          {
35              NEXT_STATE.PC=CURRENT_STATE.PC+branch_target;
36          }
37          else{
38              NEXT_STATE.PC=CURRENT_STATE.PC+4;
39          }
40          break;
```

# 3 实验结果




(a) 实验结果 1




(b) 实验结果 2

图 3.2: 实验结果

各个测试文件均得到 Simulator halted 的结果，实验验证成功。

# 4 实验总结

本次实验完成了 53 条 MIPS 指令的 C 语言实现，在给定既有框架的下实现了 MIPS 指令的五级流水的基础实现。本次实验过程中在一些指令的执行过程中未对精度溢出等问题进行判断和解决，属于本次实验的一些不足之处。本次实验代码及报告已上传到Git 仓库当中。